A case study of C source code verification: the Schorr-Waite algorithm

Thierry Hubert Claude Marché
PCRI — LRI (CNRS UMR 8623) — INRIA Futurs — Université Paris 11
Bât 490, Université Paris-sud, 91405 Orsay cedex, France
{hubert,marche}@lri.fr

Abstract

We describe an experiment of formal verification of C source code, using the CADUCEUS *tool. We performed a full formal proof of the classical Schorr-Waite graph-marking algorithm, which has already been used several times as a case study for formal reasoning on pointer programs. Our study is original with respect to previous experiments for several reasons. First, we use a general-purpose tool for C programs: we start from a real source code written in C, specified using an annotation language for arbitrary C programs. Second, we use several theorem provers as back-ends, both automatic and interactive. Third, we indeed formally establish more properties of the algorithm than previous works, in particular a formal proof of termination is made*[1].

Keywords: *Formal verification, Floyd-Hoare logic, Pointer programs, Aliasing, C programming language.*

> The Schorr-Waite algorithm is the first moutain that any formalism for pointer aliasing should climb.
> — Richard Bornat ([4], page 121)

1. Introduction

Using formal methods for verifying properties of programs at their source code level has gained more interest with the increased use of embedded programs, which are short programs where a high-level of confidence is required. Such embedded programs are no longer written in assembly language but rather in C (plane command control, cars, etc.) or in JavaCard^{TM} [23] (mobile phones, smart cards, etc.). To perform formal verification of C or Java programs, one faces the general issue of verification of pointer programs: *aliasing*, that is referencing a memory location by several pointers, must be taken into account.

The Schorr-Waite algorithm [22] is a graph-marking algorithm intended to be used in garbage collectors. It performs a depth-first traversal of an arbitrary graph structure (hence a structure where aliasing may occur), without using additional memory, but using the pointers in the graph structure itself as a backtracking stack. A first (non computer-aided) proof of correctness of this algorithm was given by Topor [24] in 1979, and in 1982 Morris [16] presented a semi-formal proof using a general mechanism for dealing with inductive data structures.

In 2000, Bornat [4] was able to perform a computer-aided formal proof of the Schorr-Waite algorithm using the Jape system [5]. To make the proof tractable, an old idea by Burstall [6] for reasoning on pointer programs was reused, called the 'component-as-array' trick: the heap memory is not modelled by a single large array, but by several ones, separating the memory into parts where it is statically known that no aliasing can occur between two parts. These parts correspond to differents fields of records (i.e. structures in C, or instance variables in Java). In 2003, the Schorr-Waite algorithm was used again as a case study by Mehta and Nipkow [14], this time for verification of pointer programs in the higher-logic system Isabelle/HOL. In 2003 also, Abrial [2] performed another verification of this algorithm, this time based on refinement, using the B system.

In 2004, we proposed a new verification method for ANSI C source code [9]. A prototype tool has been implemented, called CADUCEUS, freely available for experimentation [8]. Unlike formerly mentioned systems, CADUCEUS takes as input real source code in the C programming language, where specifications are given inside regular C comments, in the spirit of the Java Modeling Language [11]. It is not the purpose of this paper to describe CADUCEUS internal technology, which already have been presented in another conference [9]. Roughly speaking, CADUCEUS is a verification condition generator, from programs annotated with pre- and post-conditions. Those conditions are generated using a weakest precondition calculus and a monadic translation to functional programs. An original feature of CADUCEUS is its multi-prover output: the user may choose

1 This research was partly supported by the "projet GECCOO de l'ACI Sécurité Informatique", `http://gecco.lri.fr`, the AVERROES RNTL project `http://www-verimag.imag.fr/AVERROES/`, CNRS & INRIA

to establish the validity of the generated conditions with several different provers, both automatic and interactive ones. In this paper, we illustrate the use of the Simplify [1] automatic prover and the Coq [7] proof assistant.

To demonstrate that CADUCEUS is a powerful approach (and tool) to formal verification of C source, according to Bornat's citation, we performed again a verification of the Schorr-Waite algorithm, and this article reports on this experiment. With CADUCEUS, we indeed formally proved more properties of the Schorr-Waite algorithm than previous studies. The first additional property we show is the absence of *threats* (null pointer dereferencing, out-of-bounds array access, or more generally dereferencing a pointer that does not point to a regularly allocated memory block): if the initial graph structure is regularly allocated, then no invalid pointer deferencing occurs during execution of the algorithm. Even if this is not hard to do on this example, it is a crucial property to prove on C programs (in particular buffer overflow is known to be a major source of bugs), and it is indeed mandatory with the CADUCEUS approach. The second additional property we show is an extra behavioral property: everything outside the graph structure remains unchanged. The third additional property is the termination: informal termination arguments were known since Topor [24], and Abrial [2] has a termination proof integrated in the B refinement process, but we are the first to provide a formal proof of termination on the implementation itself.

This paper is organized as follows. Section 2 gives an overview of the CADUCEUS tool. In Section 3, we describe the Schorr-Waite algorithm and give it a behavioral specification in the CADUCEUS specification language. Then we describe the formal proof in Section 4, which amounts to add more annotations in the source code: a loop invariant and a variant. We conclude in Section 5 with a comparison to related works.

2. Overview of the CADUCEUS approach

In the following, we assume the reader fairly familiar with ANSI C [10]. In the CADUCEUS setting, the properties that may be checked are of two kinds. First, the program may be checked to be free of *threats*, that are operations which may lead to abnormal program termination: division by zero and such, including the crucial case of dereferencing a pointer that does not point to a valid memory block (i.e. on the current stack or allocated on the heap by a `malloc`-like call). In our approach, out-of-bounds array access is a particular case of pointer dereferencing threat, because we support full pointer arithmetic. The second kind of properties which can be proved are user-defined behavioral properties given as functions's *postconditions*.

These kinds of properties are not independent since they both usually need insertion of appropriate annotations (functions's *preconditions*, *global invariants*, *loop invariants*, etc.) as usual in a Hoare logic framework. In practice, these annotations are inserted in the source code as comments of a specific shape `/*@...*/`. The specification language we use in those comments is largely inspired by the Java Modeling Language (JML) [11]. Description of this annotation language is not the purpose of this article [8, 9], we simply introduce here what we need for the case study. One feature of this annotation language, shared with JML, is that annotations follow a syntax similar to C, so that a C programmer may learn it quite easily.

Once a C program is annotated, verification is done by running CADUCEUS on its source code in a way similar to a classical compiler, but resulting in the generation of so-called *verification conditions*: first-order predicate logic formulas whose validity implies the soundness of the program with respect to the absence of threat and to the behavioral properties given as annotations. At this point, a general purpose theorem prover must be used to establish those verification conditions. An original CADUCEUS feature is its independence with respect to the prover. It currently supports Coq [7] and PVS [17] interactive proof assistants, and Simplify [1], haRVey [18] and CVC-lite [3] automatic provers. Coherence between all prover outputs is obtained by first producing each verification condition in a common logical setting: first-order logic with equality and arithmetic on integers and reals, which can be syntactically embedded into the logic of each of the former provers. Switching from a prover to another is trivial, but on the other hand there is no way to make several provers cooperate on solving the same goal (we come back to this issue in Section 5). This means also that a new prover may easily be added via a suitable pretty-printer, as soon as it supports first-order quantification, built-in equality, and arithmetic.

To perform proof of the verification conditions with an interactive prover, the user needs to understand the modeling of the memory heap provided by CADUCEUS. The 'component-as-array' trick of Burstall and Bornat is reused, extended to arrays and pointer arithmetic in general [9]. For this case study, no pointer arithmetic occurs in the considered source code, and fortunately the generated verification conditions are exactly what they could be expected with the original 'component-as-array' modeling. That is, the extension of this trick to pointer arithmetic does not add any overhead when the source code makes no use of pointer arithmetic.

3. The Schorr-Waite algorithm

The Schorr-Waite algorithm performs a depth-first traversal of a directed graph, starting from a specific node of the graph called the root. Its main characteristic is that it

```
void schorr_waite(node root) {
  node t = root; node p = NULL;
  while (p != NULL || (t != NULL && ! t->m)) {
    if (t == NULL || t->m) {
      if (p->c) { /* pop */
        node q = t; t = p; p = p->r; t->r = q;
      }
      else { /* swing */
        node q = t; t = p->r; p->r = p->l; p->l = q; p->c = 1;
      }
    }
    else { /* push */
      node q = p; p = t; t = t->l; p->l = q; p->m = 1; p->c = 0;
    }
  }
}
```

Figure 1. C version of the Schorr-Waite algorithm

directly uses the pointers of the graph to implement backtracking. This is why its soundness is difficult to establish.

3.1. The C source code

In the version considered here, as in [4, 14], we assume that the nodes have at most two children[2]. The case of two children is the situation in a garbage collector of a pure Lisp interpreter, where the only memory allocated value is the *cons* of lists. In C, these *cons* structures can be defined as

```
typedef struct struct_node {
  unsigned int m:1, c:1; /* booleans */
  struct struct_node *l, *r;
} * node;
```

where l and r are respectively pointers to the left and to the right child (they can be set to NULL). Fields m and c are integers over 1 bit, to encode booleans. The field m is a mark: initially, all nodes of the graph will be assumed unmarked, and at the end of the traversal, they will have to be all marked. The field c is used internally, to denote which of the children is currently being explored.

The Schorr-Waite graph-marking algorithm, written directly in true C source code, is given in Figure 1. The beginning of a sample execution is illustrated in Figure 2, with each possible move 'push', 'swing' and 'pop'. The black nodes represent marked nodes, whereas white nodes are unmarked. The sample graph shown is a tree for readibility, but of course any graph could be given. t is the next node to be explored and p is the head of the backtracking stack. 'push' marks a new node and then explores the left child. 'swing' occurs when the left child has been explored: the search continues to the right child. 'pop' is used when the right child has been explored: the search goes back up.

2 extension to an arbitrary number of children is essentially the same, provided that this number is bounded by a value N ans that the c field is large enough to represent N different values

3.2. Formal specification of the algorithm

The first step in this formal verification process is to give a formal specification to the `schorr_waite` function. The informal specification says that every node in the graph, reachable from the root node, must be marked. Moreover the graph structure must be restored to its initial state.

So it appears immediately that to formally specify the algorithm, one needs to talk about reachability of some node from another in the graph. Indeed reasoning about reachability is the main part of this verification, as already noticed by previous studies of the Schorr-Waite algorithm.

At this point, the CADUCEUS methodology is quite different from the JML one: in JML, one would need to provide a so-called *pure* (i.e. side-effect free) method defining reachability and using it in annotations. In the CADUCEUS methodology, such a logical notion must be declared, by a `predicate` annotation, and supposed to be defined or axiomatized later. This is very similar to a forward declaration of a C function, whose implementation will be given later: such predicates have to be 'implemented' later, either by giving additional CADUCEUS annotations (such as `axioms`) or in the back-end prover. For the reachability predicate, we write

```
/*@ predicate
  @   reachable(node p1, node p2)
  @   reads p1->l,p1->r */
```

This declares a binary predicate on nodes, but nothing yet is said about its semantics. The `reads` clause that follows the predicate declaration above gives information on which data

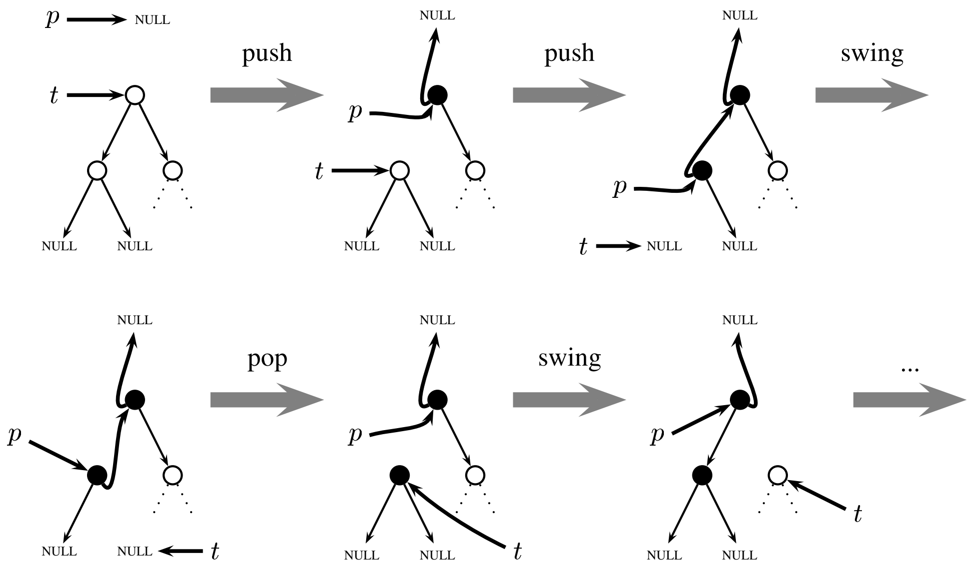

Figure 2. Schorr-Waite algorithm: sample execution

```
/*@ requires
  @   \forall node x;
  @       x != \null && reachable(root,x) => \valid(x) && ! x->m
  @ ensures
  @   (\forall node x; \old(x->l) == x->l && \old(x->r) == x->r)
  @ &&
  @   (\forall node x; x != \null && reachable(root,x) => x->m)
  @ &&
  @   (\forall node x; !reachable(root,x) => x->m == \old(x->m))
  @*/
void schorr_waite(node root) {
  ...
}
```

Figure 3. Specification of Schorr-Waite algorithm in the CADUCEUS syntax

the predicate depends on: in this case, it does depend on `p1` and `p2` but also on the children of `p1`. This may be confusing at first, because in fact this predicate depends not only on the children of `p1`, but also the grand children and so on. To really understand the meaning of the `reads` clause, one needs to remember that we use the component-as-array modeling: this declaration indeed means that this predicate will have, on the prover side, additional 'array' arguments `l` and `r` respectively representing the left and right child of all nodes. We will come back to this in Section 4.3.

Once this `reachable` predicate is declared, it is possible to give a formal specification to the algorithm, even if we did not give the semantics of `reachable` yet. The specification, in the CADUCEUS syntax, is given in Figure 3. It is made of two clauses: the `requires` clause specifies the pre-condition whereas the `ensures` clause specifies the post-condition. Both are followed by a logical formula, where logical connectives follow a C-like syntax: C operator `&&` denotes conjunction, `!` denotes negation, etc. Additional syntax is introduced to denote implication by =>, and universal quantification by `\forall` *type* x; *formula*. C expressions can be used in logical formulas as long as they are side-effect free. On this example we use equality and inequality (== and `!=`) and field accesses like `x->l`. Additional predicates exist, all prefixed by a backslash, such as `\null` which denotes the null pointer. Notice that our annotation language inherits from C the fact that booleans are not clearly distinguished from the integers: when an integer expression e is used as a logical atom, then it should be understood as the boolean $e \neq 0$, so that one can equivalently write `! x->m` or `x->m == 0`. Notice that we use a standard first-order logic: every function is total (like in

JML, i.e. no 3-valued logic) so that an expression `p->f` is defined even if `p` is `null`, but no property of it is known (very similarly to division by zero for example).

In the pre-condition, the built-in predicate `\valid`(p) means that the pointer p points to a correctly allocated memory block, i.e. that dereferencing p is safe. So the pre-condition specifies that for any node x that is not null and reachable from the root, it should be regularly allocated and its m field must be false (that is x is unmarked).

The post-condition is a conjunct of three assertions. The built-in construct `\old`(e) in CADUCEUS, inspired from the JML one, denotes the value of expression e before the execution of the function. So the first assertion means that for all nodes (even the non-reachable ones), the children are the same after the run of the algorithm, as they were before. This is of course very important to specify, because during the execution the children are modified, so one wants to prove that the initial graph structure is restored. The second assertion specifies that any non-null node x reachable from the root is now marked. So this specifies that the whole graph has been traversed. The third assertion says that for the nodes which were not reachable, their mark is not changed by the algorithm, that is the unreachable nodes are not traversed by the algorithm.

With respect to the formal specification given by Bornat [4] or the similar one in Isabelle/HOL [14], we added two things: first the pre-condition that all pointers of the graph are regularly allocated (which is the meaning of `\valid`) at the beginning, so that we are now able to prove that no wrong pointer dereferencing can occur. Second, we also talk about the non-reachable nodes of the graph: we are also able to show that they are not traversed and not modified. In Section 4.4, we add more annotations in order to establish the termination of the `schorr_waite` function.

4. Verification

The `schorr_waite` C function is now formally specified. The hard work begins: we need to prove that the implementation of Figure 1 indeed satisfies this specification. If one runs the CADUCEUS tool on that annotated code, it generates verification conditions that are actually not provable: as usual with a Hoare-like system, when there are loops like the `while` loop, it is mandatory (except in simple cases) to manually add a *loop invariant*, that is an assertion preserved by each iteration of the loop. The design of a suitable loop invariant is the most difficult part of this verification process, but suitable loop invariants were already proposed. However, some difficulties remain, the main one is that we need to express this loop invariant in the CADUCEUS syntax, which is limited to a first-order logic. Hence, the higher-order specifications of [14] are not allowed by CADUCEUS.

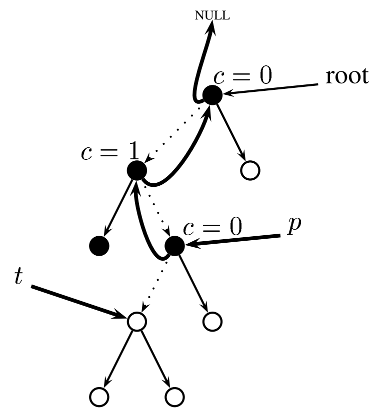

Figure 4. The backtracking stack inside the graph structure

4.1. Designing the loop invariant

The key idea for the loop invariant is that even if the graph structure is temporarily modified, the reachability of nodes is preserved. More precisely, each node that is initially reachable from the root will be always reachable either from p or from t. We also need to describe as a loop invariant how the backtracking stack is implemented using the pointers of the graph. On Figure 4, we display some state of the graph as it may occur at an arbitrary iteration of the loop. The dotted arrows correspond to the children pointers as they were at the beginning, whereas the thick arrows show their current values. The thin arrows correspond to unmodified links. The backtracking stack is indeed the list of pointers, starting from p, following either the left child when c is 0, or the right child when c is 1.

The loop invariant as it is written in the CADUCEUS annotation language is displayed on Figure 5. For specifying this loop invariant, in particular the stack, we need to talk about lists of pointers in annotations. With CADUCEUS, such a datatype from the logical side can be imported, implicitly. Then new predicates and logical function symbols can be declared, as the `reachable` predicate already introduced, which can take as arguments either C types or logical types. Here we declare

```
//@ logic plist cons(node p, plist l)
//@ predicate in_list(node p, plist l)
/*@ predicate
  @   pair_in_list(node p1, node p2,
  @                plist l) */
```

which first introduces the logical type `plist` for finite lists of pointers, then declares a logical function symbol `cons` which is intended to add an element to a list, a predicate `in_list` which is supposed to test

```
void schorr_waite(node root) {
  node t = root; node p = NULL;
  /*@ invariant
    @  (I1 :: \forall node x;
    @    \old(reachable(root,x)) => reachable(t,x) || reachable(p,x))
    @ &&
    @  (I2 :: \forall node x; x != \null =>
    @         (reachable(t,x) || reachable(p,x)) => \old(reachable(root,x)))
    @ &&
    @  (I3 :: \forall node x;
    @       ! \old(reachable(root,x)) => x->m == \old(x->m))
    @ &&
    @  \exists plist stack;
    @    (I4a :: clr_list (p,stack))
    @  &&
    @    (I4b :: \forall node p; in_list (p,stack) => p->m)
    @  &&
    @    (I4c :: \forall node x; \valid(x) && \old(reachable(root,x)) && !x->m =>
    @          unmarked_reachable(t,x) ||
    @          (\exists node y; in_list(y,stack) && unmarked_reachable(y->r,x)))
    @  &&
    @    (I4d :: \forall node x;
    @             !in_list(x,stack) => (x->r == \old(x->r) && x->l == \old(x->l)))
    @  &&
    @    (I4e ::
    @      \forall node p1; \forall node p2;
    @        pair_in_list(p1,p2,cons(t,stack)) =>
    @            (p2->c => \old(p2->l) == p2->l && \old(p2->r) == p1)
    @          &&
    @            (!p2->c => \old(p2->l) == p1 && \old(p2->r) == p2->r))
    @*/
  while (p != NULL || (t != NULL && ! t->m)) {
    ...
```

Figure 5. The loop invariant

whether a pointer belongs to a list, and finally a predicate `pair_in_list`(p_1,p_2,l), saying that p_1 and p_2 are two consecutive elements in a list l. As for `reachable`, these are only declarations, true definitions or axiomatizations of these functions and predicates will be given later.

The loop invariant finally requires the declaration of two predicates:

```
/*@ predicate
  @   unmarked_reachable(node p1,node p2)
  @   reads p1->r,p1->l,p1->m */
```

specifies that p_2 is reachable from p_1 by traversing only unmarked nodes; and

```
/*@ predicate
  @   clr_list(node p,plist stack)
  @   reads p->c,p->l,p->r */
```

specifies that `stack` is the list of pointers obtained from p as described on Figure 4. The precise definition of it will be given in Section 4.3.

In Figure 5, we use the CADUCEUS notation to name formulas with double-colons. The loop invariant is made of eight assertions:

- I1 means that each node which was initially reachable from root, is reachable from t or p.
- I2 means that conversely, each (non-null) node reachable from t or p was initially reachable from `root`. This is true only for non-null nodes, because `null` may be not reachable from `root` if the graph is cyclic.
- I3 means that for initially unreachable nodes, the mark remains unchanged.

I4a defines the backtracking stack as on Figure 4, using the `clr_list` predicate declared before.

I4b means that all nodes in the `stack` are marked.

I4c is the trickiest annotation: it means that any valid node, initially reachable from root, and not yet marked, is reachable by traversing only unmarked nodes, either from t or from the right child of a node in the stack.

I4d means that all nodes out of `stack` have the same children as initially.

I4e is essential for being able to 'recover' the correct children of nodes of the stack, as it can be seen again on Figure 4: if p_1 is in the stack, followed by p_2 (or if p_1 is t and p_2 is the head of the stack), then: if p_2`->c` is 1 then its left child is the same as it was initially and its right child is p_1; and if p_2`->c` is 0 then its left child is p_1 and its right child is the same as it was initially.

4.2. Verification using automatic provers

Running CADUCEUS on the annotated source code leads to twelve verification conditions. Five of them relate to the behavioral specification of the algorithm: loop invariant true when entering the loop, preservation of the loop invariant for each of the branches 'pop', 'swing' and 'push', and validity of the post-condition. The seven others are required to establish the absence of invalid pointer dereferencing: the first one asks for validity of `t` for the field access `t->m` in the condition of the `while`, and the six others similarly ask for validity of dereferencing `t->m` in the first `if`, `p->c` in the second `if`, `t->r` in the 'pop' branch, `p->l` in the 'swing' branch, `t->l` and `p->l` in the 'push' branch. Notice that some accesses do not require a validity check, because they are trivial enough to be automatically discharged, for example `p->r` in the 'pop' branch because it was preceeded by `p->c` and `p` did not change.

A first attempt can be made to solve these obligations with an automatic prover. Here we present the results with Simplify, which appears to be the best in that case. Four of the twelve obligations are solved automatically, which are the validity of the dereferencings occuring in the 'pop', 'swing' and 'push' branches. Indeed, a quick look at those obligations shows that they can be solved by equality reasoning: for example the `t->r` access in the 'pop' branch is valid because `t` is equal to the old value of `p` in the condition of the if, which is already assumed valid. It is not surprising at all that Simplify does not solve the other obligations, because until now we did not give any definition or axiomatization for the introduced predicates, in particular reachability. Indeed, to establish for example validity of `t->m` in the condition of the `while`, one needs to use the function's pre-condition saying that any non-null initially reachable node is valid. To establish that `t` is initially reachable, one can use the `I1` assertion of the loop invariant, as soon as one knows the simple fact that `t` is reachable from itself! So one thing that can be done is to add in the source an axiom for `reachable`:

```
/*@ axiom reachable_refl :
  @    \forall node p ; reachable(p,p) */
```

Indeed with that new axiom, Simplify automatically solves the three remaining obligations for validity of pointer dereferencing: we have certified the absence of threat.

At this point, any of our attempts to add axioms for solving the remaining obligations failed. We also tried our other supported automatic provers haRVey and CVC-lite, but they are even worse than Simplify. These automated provers are of course incomplete, and the obligations we have are apparently too difficult for them, so we switched to an interactive prover: the Coq proof assistant.

4.3. Verification with the Coq interactive prover

Unlike with Simplify, we have now all the power of the Coq specification language; in particular we do not need to axiomatize predicates in the C source file, but we can *define* them, and in particular for reachability, it is natural to use an inductive definition.

Due to lack of space, we do not present any Coq code here, but only approximations to ease reading. The whole development can be found from the CADUCEUS web page [8]. The first step is to define the reachability predicate in Coq. We proceed in a slightly different way than Mehta and Nipkow [14], who use a set-theoretic axiomatization: we use inductively defined predicates, which are more suitable for reasoning in Coq. We first introduce a more general predicate `path` which also makes precise the path between nodes. It is defined by three clauses:

$$
\begin{array}{l}
\texttt{path}(a,l,r,p,p,nil).\\
\texttt{path}(a,l,r,p_1,p_2,cons(p_1,s))\texttt{:-}\\
\quad \texttt{valid}(a,p_1),\texttt{path}(a,select(l,p_1),p_2,s)\\
\texttt{path}(a,l,r,p_1,p_2,cons(p_1,s))\texttt{:-}\\
\quad \texttt{valid}(a,p_1),\texttt{path}(a,select(r,p_1),p_2,s)
\end{array}
$$

the first one says that one may go from a node to itself by an empty path, the second (resp. the third) say that if there is a path s from the left (resp. right) child of p_1 to p_2 then $cons(p_1, s)$ is a path from p_1 to p_2. Then `reachable` is defined by the existence of a path (resp. `unmarked_reachable` by a path with unmarked nodes):

$$
\begin{array}{l}
\texttt{reachable}(a,l,r,p_1,p_2) := \exists s,\ \texttt{path}(a,l,r,p_1,p_2,s)\\
\texttt{unmarked\_reachable}(a,m,l,r,p_1,p_2) :=\\
\quad \exists s,\ \texttt{path}(a,l,r,p_1,p_2,s)\wedge\\
\qquad \forall p, p \in s \rightarrow select(m,p) = 0
\end{array}
$$

```
/*@ logic Length weight(node p, node t)
                     reads p->m,p->c,p->l,p->r  */
/*@ predicate reachable_elements(node root, plist s)
                     reads root->l,root->r */
/*@ requires
       \exists plist s; reachable_elements(root,s) &&
     ... */
void schorr_waite(node root) {
  node t = root; node p = NULL;
  /*@ invariant ...
    @ variant weight(p,t) for order_mark_m_and_c_and_stack
    @*/
  while (p != NULL || (t != NULL && ! t->m)) {
   ...
```

Figure 6. Additional annotations for termination

Notice that it is now easy to establish the validity of the axiom `reachable_refl`. It may be surprising at first that the `reachable` predicate defined here has five arguments instead of two, as declared in Section 3.2. This is the effect of the `reads` clause: as already said, this is due to the component-as-array modeling: the `reachable` predicate has two arguments in C, but in the modeling it is given extras arguments a, l, r which are arrays indexed by pointers (with $select$ as their access function). a corresponds to an allocation table which tells for each pointer whether it is allocated or not, and l, r correspond to fields of the `node` structure. It is clear that to perform the verification of a C program with CADUCEUS and an interactive prover as back-end, one needs to learn more on this modeling [8, 9].

The next step is to deal with the type `plist` used to model the stack: we simply use the finite list datatype of the Coq standard library. The `clr_list` predicate can be defined inductively with the Prolog-like clauses:

$$\begin{array}{l}\texttt{clr\_list}(a,c,l,r,p,nil).\\ \texttt{clr\_list}(a,c,l,r,p,cons(p,s))\texttt{:-}\\ \qquad\texttt{valid}(a,p),\texttt{clr\_list}(a,next(c,l,r,p),s).\end{array}$$

where

$$next(c,l,r,p) := \texttt{if}\ select(c,p)=0\ \texttt{then}\ select(l,p)\ \texttt{else}\ select(r,p)$$

A few lemmas are proved before proving the obligations, the main one, reused several times in the proofs, is to show that when p_2 is reachable from p_1, there is a path which has no cycle:

$$\forall a,l,r,p_1,p_2,s,\ \texttt{path}(a,l,r,p_1,p_2,s) \rightarrow \exists s', s' \subseteq s \wedge \texttt{no\_rep}(s') \wedge \texttt{path}(a,l,r,p_1,p_2,s')$$

where `no_rep` is a simple predicate on a list which says that no element occurs twice.

With those definitions and lemmas, we are able to manually complete all the proof obligations in Coq. We give a few numbers at the end of the next section.

4.4. Termination

Proving the termination of the algorithm is proving the termination of the while loop. In CADUCEUS, one can annotate a loop by a *variant*: an expression which decreases with respect to some well-founded ordering. It is not hard to find such a measure for the Schorr-Waite algorithm. There is a triple of natural numbers which decreases lexicographically at each iteration: the number of unmarked reachable nodes; the number of reachable nodes which have $c = 0$; the length of the stack. There is a minor issue: in our modeling, there is no assumption on the finiteness of the memory, so indeed nothing says that the set of reachable nodes is finite. So this is an extra assumption we have to insert in the pre-condition. The additional annotations for termination are given in Figure 6, where in Coq we define

$$\texttt{reachable\_elements}(a,l,r,root,s) := \forall p, p \in s \leftrightarrow \texttt{reachable}(a,l,r,root,p)$$

The `weight` function and the `order_mark_m_and_c_and_stack` relation are also defined in Coq according to the informal definition of the measure above.

With the completed annotated code, CADUCEUS generates two additional verification conditions. The first is to show that the ordering is well-founded, whereas the second is a specificity of the CADUCEUS verification condition generator, trivially solved by equality reasoning. The verification that the measure decreases at each iteration is added to the obligations corresponding to preservation of the loop invariant.

The well-foundedness obligation is not passed to automatic provers like Simplify because it is a second-order formula. So finally, the Simplify prover solves 8 of the 13 obligations, which is a quite good score but of course the 5 remaining obligations are by far the most difficult.

With Coq, we completed all the obligations including the well-foundedness. Here are a few numbers about the whole Coq proof: definition of predicates on lists, reachability and such, and corresponding lemmas amount to 317 lines of definitions and 589 lines of proof tactics. The generated verification conditions amount to 985 lines, and 2411 lines of tactics were inserted manually to prove them. Approximately 40% of the work was devoted to the termination proof, both in terms of lines of Coq text and of time spent: proving the obligations without termination required around 3 weeks, and the termination proof required 2 more weeks. These times include several iterations of the process of modifying the annotations, regenerating the verification conditions and modifications of the proof scripts.

These numbers may seem quite big for a program of only 10 lines of C code. But of course this algorithm is especially clever, and it should not be surprising that its justification is hard. Another reason is that we had to define in Coq many preliminary definitions and lemmas (regarding reachability) from scratch: we could have avoided this if Coq had an existing rich library for graphs. Indeed, some of our preliminaries will be included in the next version of Coq. Compared to the similar proof in Isabelle/HOL [14], we have roughly three times more lines of proofs: we believe that this is due to the weakness of automatic tactics of Coq (but also remember that we have the additional proofs of termination and absence of threats).

5. Conclusion

We presented a machine-checked full verification of the Schorr-Waite algorithm, directly on a real C source version of it. We proved the behavioral properties of it, like previous works [4, 14, 2], but we have also proved that no invalid pointer dereferencing may occur, and the termination.

The termination does not use any unique feature of CADUCEUS: termination is indeed established in [2] as a condition for refinement, and we believe that it could probably also proved by others as well. The important point is that termination is smoothly integrated in our methodology, thanks to the powerful clause `variant`, which allows the use of an arbitrary well-founded relation.

Regarding the absence of invalid pointer dereferencing, it has to be said that advanced static analysis techniques are able to establish it also. The Schorr-Waite algorithm is one of the examples automatically handled by the TVLA approach [21, 19], where a powerful analysis of reachability in pointer structures is done. But until now, as far as we know, only our approach is able to handle both the pointer dereferencing checks and the validity of behavioral properties given by the user. Notice that with CADUCEUS, as shown in Section 4.2, absence of invalid pointer dereferencing is established automatically, even if CADUCEUS does not do any kind of reachability analysis by itself.

Finally, we emphasize again that we use a generic tool for arbitrary C programs, without anything specialized for this case study, which additionally gives the choice of the back-end prover. At the beginning, it was not clear whether the CADUCEUS specification language would be enough, in particular because it is first-order, but we finally succeeded. We claim this is a very successful case study, which provides a clear evidence that the CADUCEUS methodology [9] is powerful.

Since we proposed a specification language inspired from JML, and since we also develop a similar tool [12, 13] for Java programs annotated in JML, it would be natural to perform the same case study on an Java/JML version of the Schorr-Waite algorithm. We already made attempts into this direction, indeed we tried first the in-place list reversal algorithm of Bornat [4, 9]. However we met some problems due to small differences between JML and our specification language. First, where we use predicates in annotations which are only defined on the prover side, one should use model methods in JML, but those have to be defined (or axiomatized) in JML itself, which is not obvious to do for inductively defined predicates like `reachable` or `clr_list`. Second, JML has some limitations, such as forbidding the use of `\old` in loop invariants (this limitation has notified to the JML mailing list, and is expected to be relaxed in a near future). So we expect to make the similar case study on a JML-annotated JAVA program in a near future.

There are still many potential improvements in our approach. One improvement clearly emphasized by this case study is the ability to perform proofs with *cooperation* between automatic provers and interactive ones. Interactive provers are nice because they are very expressive and allow to prove manually difficult goals, but their automatic tactics are less powerful than those of automatic provers. This is especially true for Coq, because any decision procedure in it needs to provide a checkable proof trace, making its integration very difficult. So one of the major goals of our research is now to develop a platform for verification where automatic theorem proving will be used as far as possible, and only when this fails the user would be asked for manual interaction. Since CADUCEUS also supports the interactive prover PVS, making the proofs with PVS, which is known to have more efficient automatic reasoning than Coq, could be a good alternative (but unfortunately we are not experienced PVS users ourselves).

This case study was motivated by Bornat's assertion given at the beginning. But the Schorr-Waite program is not representative of the kind of C program we are really interested in. We focus in particular on embedded industrial

C programs, where a high level of confidence is required: CADUCEUS is currently under experimentation at Axalto (smart cards) and Dassault Aviation (aeronautics) companies. For this goal, several improvements to the CADUCEUS tool are in progress. There are scaling up issues for dealing with code of large size. Automatic annotation of programs is an important need. Properties of separation of data appear to be crucial for large programs: we are currently investigating improvements, in particular by integration of advanced static analysis techniques, and ideas from separation logic [20]. Generally speaking adding static analysis techniques to our setting is a major goal.

Among more academic future work, we wonder what would be the next "mountain to climb" for pointer program verification? We think that memory allocation and multi-procedure programs (where specification of separation is crucial) are important issues. We currently working on advanced graph algorithms (such as Dijkstra's shortest path [15] and planarity test) and Ukkonen's algorithm [25].

Acknowledgments. We thank Jean-Christophe Filliâtre for his fruitful help, advices and remarks about this case study.

References

[1] The Simplify decision procedure (part of ESC/Java). `http://research.compaq.com/SRC/esc/simplify/`.

[2] J.-R. Abrial. Event based sequential program development: Application to constructing a pointer program. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805, pages 51–74. Springer-Verlag, 2003.

[3] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In R. Alur and D. A. Peled, editors, *16th International Conference on Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518, Boston, MA, USA, July 2004. Springer-Verlag.

[4] R. Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[5] R. Bornat and B. Sufrin. Animating formal proof at the surface: The Jape proof calculator. *The Computer Journal*, 42(3):177–192, 1999. `http://jape.org.uk`.

[6] R. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

[7] The Coq Proof Assistant. `http://coq.inria.fr/`.

[8] J.-C. Filliâtre, T. Hubert, and C. Marché. The Caduceus tool for the verification of C programs. `http://why.lri.fr/caduceus/`.

[9] J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. In J. Davies, W. Schulte, and M. Barnett, editors, *Sixth International Conference on Formal Engineering Methods*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29, Seattle, WA, USA, Nov. 2004. Springer-Verlag.

[10] B. Kernighan and D. Ritchie. *The C Programming Language (2nd Ed.)*. Prentice-Hall, 1988.

[11] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106, 2000.

[12] C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004. `http://krakatoa.lri.fr`.

[13] C. March and C. Paulin-Mohring. Reasoning about Java programs with aliasing and frame conditions. In J. Hurd and T. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science. Springer-Verlag, Aug. 2005.

[14] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *19th Conference on Automated Deduction*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[15] J. S. Moore and Q. Zhang. Proof pearl: Dijkstra's shortest path algorithm verified with ACL2. In J. Hurd and T. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science. Springer-Verlag, Aug. 2005.

[16] J. M. Morris. A proof of the Schorr-Waite algorithm. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 25–51. Reidel, 1982. Proceedings of the 1981 Marktoberdorf summer school.

[17] The PVS system. `http://pvs.csl.sri.com/`.

[18] S. Ranise and D. Deharbe. Light-weight theorem proving for debugging and verifying units of code. In *Proc. SEFM'03*, Canberra, Australia, Sept. 2003. IEEE Computer Society Press. `http://www.loria.fr/equipes/cassis/softwares/haRVey/`.

[19] T. W. Reps, S. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In P. Degano, editor, *12th European Symposium on Programming, ESOP 2003*, volume 2618 of *Lecture Notes in Computer Science*, pages 380–398. Springer-Verlag, 2003.

[20] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *17h Annual IEEE Symposium on Logic in Computer Science*. IEEE Comp. Soc. Press, 2002.

[21] M. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Prog. Lang. Syst.*, 24(3):217–298, 2002.

[22] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM*, 10:501–506, 1967.

[23] Sun Microsystems. The JavaCard™ application programming interface (API). `http://java.sun.com/products/javacard/`.

[24] R. W. Topor. The correctness of the Schorr-Waite list marking algorithm. *Acta Inf.*, 11:211–221, 1979.

[25] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14:249–260, 1995.